CS25100: Data Structures and Algorithms, Spring 2017

Project 3: Password Cracking

In this project you will create a C++ program that uses a symbol table to break a password encoding scheme.

Description

When a system's login manager is presented with a password, it needs to check whether that password corresponds to the user's name in its internal tables. The naive method would be to store passwords in a symbol table with the users' names as keys, but that method is vulnerable to someone getting unauthorized access to the system's table which would expose all of the users' passwords. Instead, most systems use a more secure method where the system keeps a symbol table that stores an encrypted password for each user. When a user types a password, that password is encrypted and checked against the stored value. If it matches, the user is allowed into the system.

For it to be effective, this scheme requires an encryption method with two properties: encrypting a password should be easy (since it has to be done each time a user logs in), and recovering the original password from the encrypted version should be hard.

Subset-sum Encryption

A simple method for managing passwords works as follows: The length of all passwords is set to a specific number of bits, say N. The system maintains a table T of N integers that are each N bits long. To encrypt a password, the system uses the password to select a subset of the numbers in T and add them together: the sum (modulo 2^N) is the encrypted password.

The following tiny example for 5-bit keys illustrates the process. Suppose that the table has the following five 5-bit numbers:

```
0. 10110     0
1. 01101     0
2. 00101     1
3. 10001     0
4. 01011     1
```

For this table, the password `00101` would be encrypted as `10000`, since it says to use the sum of rows two and four in the table, and `00101 + 01011 = 10000`. Of course, in practice, the table would be much bigger, as discussed below.

Now, suppose you get access to the system table T and you also capture the user names and the corresponding encrypted passwords. This information does not get you into the system: to crack a password, you need to know a subset of T that sums to a given encrypted password. The security of the system depends on the difficulty of this problem (which is known as the *subset sum* problem). Obviously, the password length has to be set long enough to stop you from trying all possibilities, but short enough so that users can remember and type the passwords. In this assignment, you will see that passwords need to be longer than you might think. With N bits there are 2^N different subsets, so it would seem that 40 or 50 bits should be enough, but that is not the case.

Details

Rather than using numbers, it is typical to use some convenient translation from what users type to numbers. For this assignment, we use an alphabet of 32 characters (lowercase letters plus first six digits) in passwords, and encode them as arrays of 5-bit integers (`char`s) with 0 encoding `'a'`, 1 encoding `'b'`, and so forth. Thus, N = 5*C where C is the number of characters in the password.

To get started, you can use the code in `encrypt.cpp`, which is the code that the system administrator would use to encrypt a user's password. It reads in the table T, then uses the bits of the password to select the words from the table to add together, and prints out the encrypted version. You can also use the tables `easy5.txt`, `easy8.txt`, `rand5.txt`, and `rand8.txt`. The first and third are for 5-`char` (25-bit) keys; the second and fourth are for 8-`char` (40-bit) keys.

For example, with the constant `C` defined to `8` in the code, you should get the following result for the password `password`. The program prints out a proof that the encryption works, for your use in understanding the process:

```
% g++ -std=c++11 -DUSE_8_CHARS -o encrypt encrypt.cpp
% ./encrypt password rand8.txt --verbose
   password  15  0 18 18 22 14 17  3   0111100000100101001010110011101000100011

 1 gobxmqkt   6 14  1 23 12 16 10 19   0011001110000011011101100100000101010011
 2 qdrvjxwz  16  3 17 21  9 23 22 25   1000000011100011010101001101111011011001
 3 joobqxtz   9 14 14  1 16 23 19 25   0100101110011100000110000101111001111001
 4 xnoixmnk  23 13 14  8 23 12 13 10   1011101101011100100010111011000110101010
10 tcixtvem  19  2  8 23 19 21  4 12   1001100010010001011110011101010010001100
13 lqtsdtca  11 16 19 18  3 19  2  0   0101110000100111001000011100110001000000
15 zlptzlfp  25 11 15 19 25 11  5 15   1100101011011111001111001010110010101111
18 gmjuvyqw   6 12  9 20 21 24 16 22   0011001100010011010010101110001000010110
20 uoqrdhwp  20 14 16 17  3  7 22 15   1010001110100001000100011001111011001111
22 ltdkzndz  11 19  3 10 25 13  3 25   0101110011000110101011001011010001111001
23 btezrznq   1 19  4 25 17 25 13 16   0000110011001001100110001110010110110000
26 bujilqno   1 20  9  8 11 16 13 14   0000110100010010100001011100000110101110
27 qgaicljl  16  6  0  8  2 11  9 11   1000000110000000010000001001011010010101 1
28 yyefwcld  24 24  4  5 22  2 11  3   1100011000001000010110110000100101100011
30 gnvowyjk   6 13 21 14 22 24  9 10   0011001101101010111010110110000100101010
34 aynzobxh   0 24 13 25 14  1 23  7   0000011000011011100101110000011011100111
38 lxwewfhh  11 23 22  4 22  5  7  7   0101110111101100010010110001010011100111
39 aenipbjd   0  4 13  8 15  1  9  3   0000000100011010100001111000010100100011
   --------  -----------------------   ----------------------------------------
   vbskbezp  21  1 18 10  1  4 25 15   1010100001100100101000001001001100101111
```

Overview of Tasks

This project is divided into two programming tasks. Make sure you read and understand the given `*.hpp` and `*.cpp` files as this will help you grasp what is already given and what needs to be implemented. You will **not** be adding any new files to the project, you should only modify the existing `*.hpp` and `*.cpp` files available on Vocareum.

Task 1: Brute Force Solution

Your first task is to write a brute force decrypt program `brute.cpp` that does the opposite of `encrypt.cpp`. Given an encrypted password and the table, it should "crack" (find) the original password. You should be able to easily break 5-char passwords, since there are only 32*32*32*32*32 (about 33 million) different possible subsets. Breaking 6-char passwords will take longer. Note that the number of characters **C** to be used by your program will be determined at compile time. More on this below.

Modify the C++ class called **`Brute`**. This must comply with the following very simple interface:

- **`Brute(const std::string& table_filename)`:** Initialize the class with the table contained in **`table_filename`**.
- **`void decrypt(const std::string& key_to_decrypt)`:** Use a brute force algorithm to decrypt the encrypted key **`key_to_decrypt`** and print out the solutions found.

You also need to write a main function:

- **`int main(int argc, char *argv[])`:** Read in an encrypted password and the name of a file containing the table to b used and decrypt it using the brute force algorithm.

To compile and run your brute force solution, you need to invoke the compile as follows:

```
% g++ -std=c++11 -O2 -DUSE_<C>_CHARS -o brute brute.cpp
% ./brute <encrypted> <table_filename>
```

where <C> is the number of characters to use. For instance, using <C>=5 in the instructions above, you should obtain:

```
% ./brute exvx5 rand5.txt
i0ocs
passw
```

Note that there may not be a unique solution, so your program should print out all of them. Note also that this approach will not be viable for longer passwords. For example, for 10-char passwords, there are 32^10 (over 1000 trillion) different possible subsets.

Task 2: Symbol Table Solution

Your next task is to write a faster decryption class **Symbol** (`symbol.cpp`) that is functionally equivalent to `brute.cpp`, but fast enough to decrypt 10-char passwords in a reasonable amount of time. To do so, use a symbol table. The basic idea is to take a subset S of the table T, compute all possible subset sums that can be made with S, put those values into a symbol table, then use that symbol table to check all those possibilities with ideally one but at most a few lookups, i.e., not exceeding *O(log N)* where *N* is the size of the symbol table.

When you consider the scheme just sketched, several questions immediately come to mind: How big should S be? Which data structure among the ones seen in class is appropriate for the symbol table? Addressing these questions is the substance of your work for this task. Note that your symbol table will store a large number of entries so you have to make sure it is memory-efficient.

Since this project is memory- and CPU-intensive, your results will vary with the machine you run it on. Should you choose to run your code outside of Vocareum, make sure to not run your simulations on heavily-loaded CS machines (e.g., `data.cs.purdue.edu` or `lore.cs.purdue.edu`). Any of the machines in the LWSN basement should be fine. You should debug your program for 5, 6-`char`, then move up to 8 and 10-

`char` passwords. Your goal, of course, is to be able to decrypt any password that was encrypted with `encrypt.cpp`, as in, for example:

```
% g++ -std=c++11 -O2 -DUSE_8_CHARS -o symbol symbol.cpp
% ./symbol vbskbezp rand8.txt
koaofbmx
password
xvbyofnz
1p1ngsgg
```

Task 3: Performance Analysis

For this task you will measure the average time it takes each program to decrypt passwords of various lengths. Specifically, you will complete the file `analysis.txt` by filling out the fields shown below. Note that you will only consider passwords of length 8 and 10 for the symbol table solution. You will also write at the bottom of each column the asymptotic complexity of the corresponding cracking algorithm in big theta notations.

```
% cat analysis.txt

Char                        Brute                         Symbol
------------------------------------------------------------
 4                            ?                              ?
 5                            ?                              ?
 6                            ?                              ?
 8                                                           ?
10                                                           ?
------------------------------------------------------------
Complexity                    ?                              ?
```

Test Cases

- To test both tasks, you can easily obtain an encrypted password string using `encrypt.cpp`. Several tables are given to you, corresponding to different password lengths.

Deliverables

You should **not** add any new files to the project. You are free to modify existing `*.hpp` and `*.cpp` files available on Vocareum.

Submission

Submit your solution before March 24 at 11:59pm. Vocareum will be used to submit assignments. Directly upload all source code used as well as the `analysis.txt`. Finally, provide a README file with your name and anything you would like us to know about your program (like errors, special conditions, etc).